**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

**МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

(национальный исследовательский университет)

«МАИ»

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра 304 «Вычислительные машины, системы и сети»

Отчёт по курсовому проекту по дисциплине

«Конструирование программного обеспечения»

на тему:

«Вычисление площади фигуры на экране»

Выполнил:

студент группы М3О-325Бк-19 Соломенцев Д.М.

Проверил: к.т.н. профессор Гагарин А.П.

Москва 2021

**Оглавление**

[Введение 3](#_Toc90633902)

[Основные подходы к решению задачи 4](#_Toc90633903)

[Формула площади Гаусса 4](#_Toc90633904)

[Алгоритм поиска замкнутых контуров 4](#_Toc90633905)

[Основные программные решения 5](#_Toc90633906)

[Работа программы 9](#_Toc90633907)

[Текст программы 11](#_Toc90633908)

[Текст основной программы 11](#_Toc90633909)

[Текст программы дополнительного окна для вычисления площади по координатам 25](#_Toc90633910)

# **Введение**

Требуется создать сервис по вычислению площади фигуры на экране на языке C# средствами Visual Studio.

Созданное приложение будет представлять собой графический редактор, в котором из всего множества проведенных линий будут формироваться фигуры и их площади будут автоматически определены. Помимо этого, будет реализована возможность вычисления площади фигуры по введенным координатам точек.

# **Основные подходы к решению задачи**

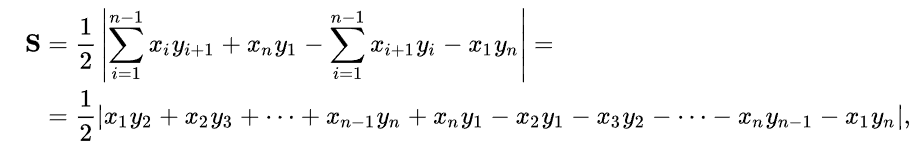
Основным в данной работе является вычисление площади фигуры, вершины которой заданы в декартовых координатах. Поэтому для решения этой задачи была использована формула площади Гаусса.

Помимо этого, был использован алгоритм поиска замкнутых контуров, составленных из прямых для определения всех возможных фигур.

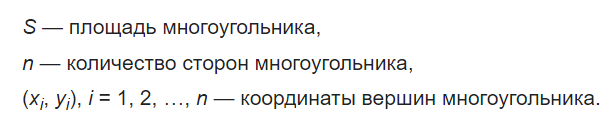
## **Формула площади Гаусса**

**Формула площади Гаусса** (формула землемера или формула шнурования, или алгоритм шнурования) — формула определения площади простого многоугольника, вершины которого заданы декартовыми координатами на плоскости. В формуле векторным произведением координат и сложением определяется площадь области, охватывающей многоугольник, а затем из нее вычитается площадь окружающего многоугольника, что дает площадь многоугольника внутри.

Формула может быть представлена следующим выражением:



где



## **Алгоритм поиска замкнутых контуров**

В работе был использован алгоритм, предназначенный для поиска всех возможных замкнутых контуров, т.е. всех возможных фигур. Сам алгоритм является немного измененным алгоритмом поиска в глубину, с тем отличием, что при спуске вглубь графа формируется список прямых, из которых в конечном итоге будет сформирована новая фигура.

# **Основные программные решения**

При добавлении на рабочую область новой прямой, она заносится в List всех прямых.

Класс прямой:

class Line

{

public Point LineStartPoint;

public Point LineEndPoint;

public Dictionary<int, Line> ConnectedLines = new Dictionary<int, Line>();

public Line(Point Start,Point End)

{

LineStartPoint = Start;

LineEndPoint = End;

}

public Line() { }

},

где

LineStartPoint – начальная точка прямой

LineEndPoint – конечная точка прямой

ConnectedLines – словарь, содержащий всех соседей текущей прямой

В случае, если новая прямая совпадает с уже существующей прямой, то она не добавится в лист всех прямых.

Когда количество прямых превысит 2, вызовется метод LineCycle(), в котором создается новый словарь, содержащий все прямые, краевые точки которых имеют хотя бы одно соединение с другими прямыми.

Dictionary<int, Line> LinesShape2 = new Dictionary<int, Line>();

foreach(var item in Lines)

{

LinesShape2.Add(Lines.IndexOf(item), item);

}

bool IsChanged=true;

while (IsChanged)

{

IsChanged = false;

foreach (var item in LinesShape2)

{

if (IsLineAlone(item.Value, LinesShape2))

{

LinesShape2.Remove(item.Key);

IsChanged = true;

break;

}

}

},

где

LinesShape2 – словарь, содержащий все прямые, краевые точки которых имеют хотя бы одно соединение с другими прямыми.

В данном алгоритме последовательно удаляются прямые, краевые точки которых не имеют хотя бы одно соединение с другими прямыми.

Когда количество прямых в новом словаре превысит 2, вызовется рекурсивный метод ListLine, который по окончании своей работы сформирует лист, который содержит все возможные замкнутые контуры, которые формируют фигуру.

private void ListLine(int CurrentLineIndex, Dictionary<int, Line> LinesShape)

{

foreach(var item in LinesShape[CurrentLineIndex].ConnectedLines)

{

if (LinesShape.ContainsKey(item.Key))

{

if (!CurrentShape.Contains(item.Key))

{

CurrentShape.Add(CurrentLineIndex);

ListLine(item.Key, LinesShape);

}

if(item.Key== StartLineIndex&& CurrentShape.Count>=2)

{

CurrentShape.Add(CurrentLineIndex);

Dictionary<int, Line> TmpDict = new Dictionary<int, Line>();

Dictionary<int, Line> ShapeDict = new Dictionary<int, Line>();

foreach (var dig in CurrentShape)

{

TmpDict.Add(dig, Lines[dig]);

ShapeDict.Add(dig, Lines[dig]);

}

bool IsCorrect = true;

bool IsShape = true;

bool IsChanged = true;

ShapeDict.Remove(ShapeDict.First().Key);

while (IsChanged)

{

IsChanged = false;

foreach (var line in ShapeDict)

{

if (IsLineAlone(line.Value, ShapeDict))

{

ShapeDict.Remove(line.Key);

IsChanged = true;

break;

}

}

}

if(ShapeDict.Count>0)

{

IsShape = false;

}

foreach (var line in TmpDict)

{

if (IsLineAlone(line.Value, TmpDict))

{

IsCorrect = false;

break;

}

}

if (IsCorrect&&IsShape)

{

List<int> tmp = new List<int>();

foreach (var dig in CurrentShape)

{

tmp.Add(dig);

}

tmp.Sort();

if (!AllShapesContains(tmp))

{

AllShapes.Add(new List<int>(tmp));

}

}

CurrentShape.RemoveAt(CurrentShape.Count - 1);

}

}

}

if (CurrentShape.Count > 0)

{

CurrentShape.RemoveAt(CurrentShape.Count - 1);

}

return;

}

Класс фигуры:

class Shape

{

public List<Line>LinesList=new List<Line>();

public double Square;

public Shape() { }

public Shape(List<Line> list)

{

foreach(var item in list)

{

LinesList.Add(item);

}

}

public void SquareCalc()

{

Square = 0;

for(int i=0;i< LinesList.Count-1;i++)

{

Square += LinesList[i].LineStartPoint.X \* LinesList[i].LineEndPoint.Y;

Square -= LinesList[i].LineEndPoint.X \* LinesList[i].LineStartPoint.Y;

}

Square += LinesList[LinesList.Count - 1].LineStartPoint.X \* LinesList[0].LineStartPoint.Y-

LinesList[LinesList.Count - 1].LineStartPoint.Y \* LinesList[0].LineStartPoint.X;

Square = Math.Abs(Square);

Square /= 2.0;

}

},

где

LinesList – лист, содержащий прямые, которые формируют замкнутый контур.

Square – площадь фигуры

# **Работа программы**

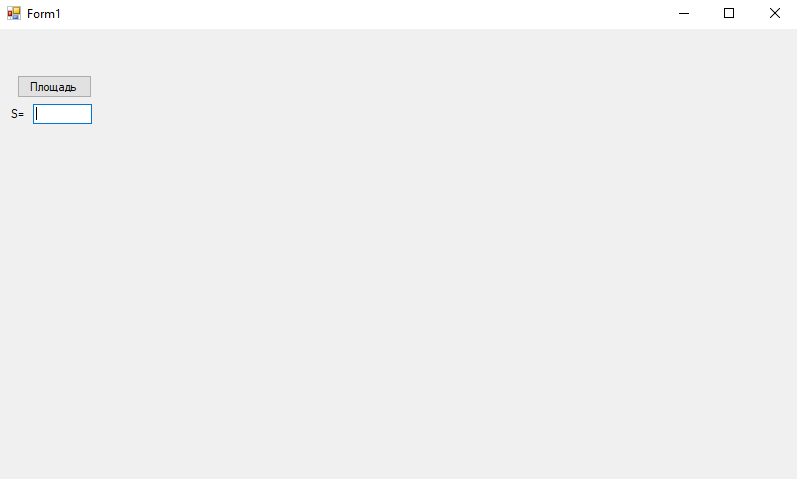


рис.1 Окно приложения при запуске

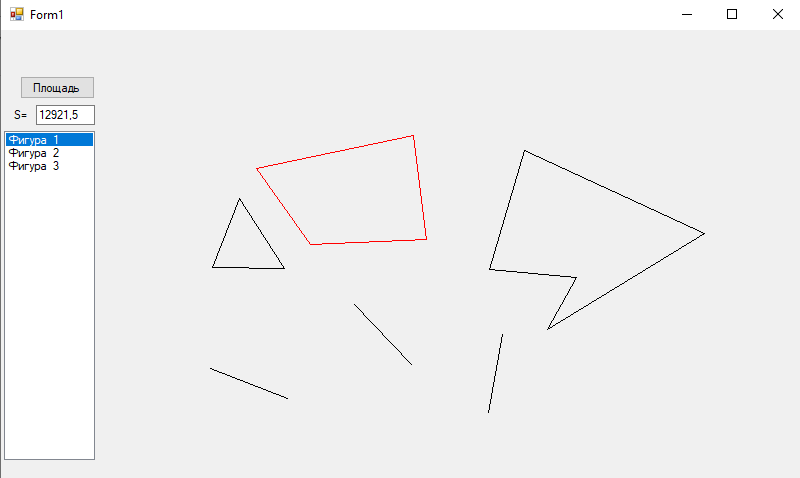


рис.2 Окно приложения в работе

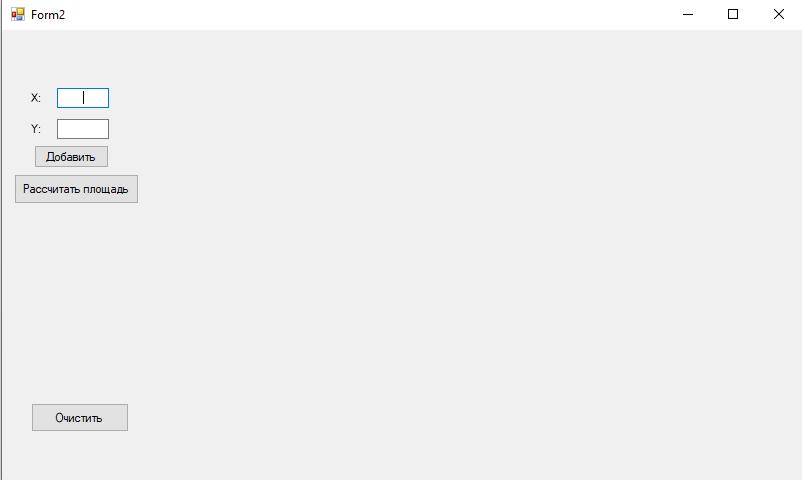


рис.3 Дополнительное окно при запуске

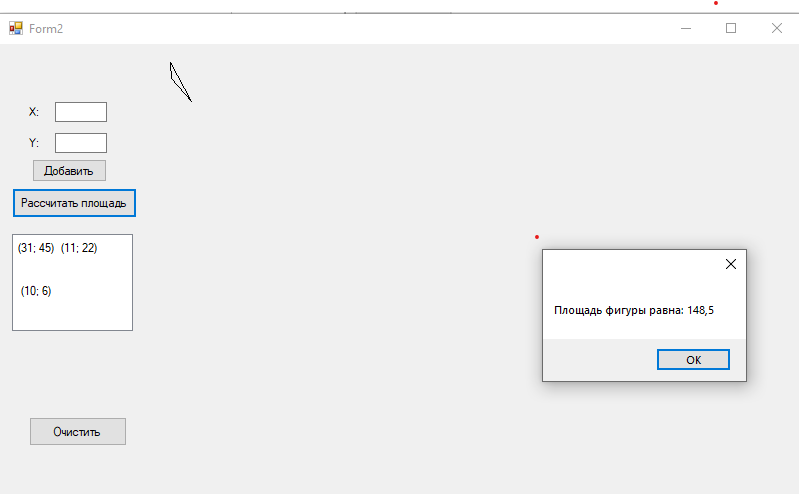


рис.4 Дополнительное окно в работе

# **Текст программы**

## **Текст основной программы**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace KPO\_Solomentsev

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

listBox1.Hide();

ToolTip tooltip = new ToolTip();

tooltip.SetToolTip(CalcButton, "Рассчитать площадь по точкам");

}

class Shape

{

public List<Line>LinesList=new List<Line>();

public double Square;

public Shape() { }

public Shape(List<Line> list)

{

foreach(var item in list)

{

LinesList.Add(item);

}

}

public void SquareCalc()

{

Square = 0;

for(int i=0;i< LinesList.Count-1;i++)

{

Square += LinesList[i].LineStartPoint.X \* LinesList[i].LineEndPoint.Y;

Square -= LinesList[i].LineEndPoint.X \* LinesList[i].LineStartPoint.Y;

}

Square += LinesList[LinesList.Count - 1].LineStartPoint.X \* LinesList[0].LineStartPoint.Y-

LinesList[LinesList.Count - 1].LineStartPoint.Y \* LinesList[0].LineStartPoint.X;

Square = Math.Abs(Square);

Square /= 2.0;

}

}

class Line

{

public Point LineStartPoint;

public Point LineEndPoint;

public Dictionary<int, Line> ConnectedLines = new Dictionary<int, Line>();

public Line(Point Start,Point End)

{

LineStartPoint = Start;

LineEndPoint = End;

}

public Line() { }

}

List<Line> LineShapesCycle = new List<Line>();

List<Shape> Shapes = new List<Shape>();

List<Line> Lines = new List<Line>();

Line CurrentLine = new Line();

bool CurrentLineClicked = false;

List<int> CurrentShape = new List<int>();

List<List<int>> AllShapes=new List<List<int>>();

int StartLineIndex=0;

private void PicturePaint(object sender, PaintEventArgs e)

{

textBox2.Clear();

Pen pen = new Pen(Color.Black);

foreach (var item in Lines)

{

e.Graphics.DrawLine(pen,item.LineStartPoint, item.LineEndPoint);

}

e.Graphics.DrawLine(pen, CurrentLine.LineStartPoint, CurrentLine.LineEndPoint);

}

private void PictureMouseDown(object sender, MouseEventArgs e)

{

CurrentLineClicked = true;

CurrentLine.LineStartPoint.X = e.X;

CurrentLine.LineStartPoint.Y = e.Y;

Point NearestP = NearestPoint(e.X, e.Y);

if (Math.Sqrt(Math.Pow((e.X - NearestP.X), 2) + Math.Pow((e.Y - NearestP.Y), 2)) < 10)

{

CurrentLine.LineStartPoint.X = NearestP.X;

CurrentLine.LineStartPoint.Y = NearestP.Y;

}

}

private void PictureMouseUp(object sender, MouseEventArgs e)

{

CurrentLineClicked = false;

bool IsAdd = true;

if(Math.Sqrt(Math.Pow((CurrentLine.LineStartPoint.X- CurrentLine.LineEndPoint.X),2)+ Math.Pow((CurrentLine.LineEndPoint.Y- CurrentLine.LineStartPoint.Y), 2))>10)

{

Lines.Add(new Line(CurrentLine.LineStartPoint, CurrentLine.LineEndPoint));

for (int i = 0; i < Lines.Count - 1; i++)

{

if (Lines[Lines.Count - 1].LineEndPoint == Lines[i].LineEndPoint && Lines[Lines.Count - 1].LineStartPoint == Lines[i].LineStartPoint ||

Lines[Lines.Count - 1].LineStartPoint == Lines[i].LineEndPoint && Lines[Lines.Count - 1].LineEndPoint == Lines[i].LineStartPoint)

{

Lines.Remove(Lines[Lines.Count - 1]);

IsAdd = false;

break;

}

}

if (IsAdd)

{

for (int i = 0; i < Lines.Count - 1; i++)

{

if (Lines[Lines.Count - 1].LineEndPoint == Lines[i].LineEndPoint || Lines[Lines.Count - 1].LineEndPoint == Lines[i].LineStartPoint ||

Lines[Lines.Count - 1].LineStartPoint == Lines[i].LineEndPoint || Lines[Lines.Count - 1].LineStartPoint == Lines[i].LineStartPoint)

{

Lines[i].ConnectedLines.Add(Lines.Count - 1, Lines[Lines.Count - 1]);

Lines[Lines.Count - 1].ConnectedLines.Add(i, Lines[i]);

}

}

if (Lines.Count > 2)

{

LineCycle();

}

}

}

}

private void LineCycle()

{

Dictionary<int, Line> LinesShape2 = new Dictionary<int, Line>();

foreach(var item in Lines)

{

LinesShape2.Add(Lines.IndexOf(item), item);

}

bool IsChanged=true;

while (IsChanged)

{

IsChanged = false;

foreach (var item in LinesShape2)

{

if (IsLineAlone(item.Value, LinesShape2))

{

LinesShape2.Remove(item.Key);

IsChanged = true;

break;

}

}

}

if (LinesShape2.Count > 2)

{

int PrevCount= AllShapes.Count;

foreach (var item in LinesShape2)

{

StartLineIndex = item.Key;

ListLine(StartLineIndex, LinesShape2);

}

if (PrevCount< AllShapes.Count)

{

listBox1.Show();

for(int i=PrevCount;i< AllShapes.Count;i++)

{

Shape shape = new Shape();

foreach (var dig in AllShapes[i])

{

shape.LinesList.Add(Lines[dig]);

}

LineSort(shape.LinesList);

Shapes.Add(shape);

shape.SquareCalc();

listBox1.Items.Add("Фигура " + " " + (i+1).ToString());

}

}

}

}

private void ListLine(int CurrentLineIndex, Dictionary<int, Line> LinesShape)

{

foreach(var item in LinesShape[CurrentLineIndex].ConnectedLines)

{

if (LinesShape.ContainsKey(item.Key))

{

if (!CurrentShape.Contains(item.Key))

{

CurrentShape.Add(CurrentLineIndex);

ListLine(item.Key, LinesShape);

}

if(item.Key== StartLineIndex&& CurrentShape.Count>=2)

{

CurrentShape.Add(CurrentLineIndex);

Dictionary<int, Line> TmpDict = new Dictionary<int, Line>();

Dictionary<int, Line> ShapeDict = new Dictionary<int, Line>();

foreach (var dig in CurrentShape)

{

TmpDict.Add(dig, Lines[dig]);

ShapeDict.Add(dig, Lines[dig]);

}

bool IsCorrect = true;

bool IsShape = true;

bool IsChanged = true;

ShapeDict.Remove(ShapeDict.First().Key);

while (IsChanged)

{

IsChanged = false;

foreach (var line in ShapeDict)

{

if (IsLineAlone(line.Value, ShapeDict))

{

ShapeDict.Remove(line.Key);

IsChanged = true;

break;

}

}

}

if(ShapeDict.Count>0)

{

IsShape = false;

}

foreach (var line in TmpDict)

{

if (IsLineAlone(line.Value, TmpDict))

{

IsCorrect = false;

break;

}

}

if (IsCorrect&&IsShape)

{

List<int> tmp = new List<int>();

foreach (var dig in CurrentShape)

{

tmp.Add(dig);

}

tmp.Sort();

if (!AllShapesContains(tmp))

{

AllShapes.Add(new List<int>(tmp));

}

}

CurrentShape.RemoveAt(CurrentShape.Count - 1);

}

}

}

if (CurrentShape.Count > 0)

{

CurrentShape.RemoveAt(CurrentShape.Count - 1);

}

return;

}

private bool AllShapesContains(List<int> CurrentShape)

{

foreach(var item in AllShapes)

{

if(item.SequenceEqual(CurrentShape))

{

return true;

}

}

return false;

}

private void LineSort(List<Line> list)

{

for(int i=0;i<list.Count-1;i++)

{

for (int j = i+1; j < list.Count; j++)

{

if(list[i].LineEndPoint== list[j].LineEndPoint|| list[i].LineEndPoint == list[j].LineStartPoint)

{

if (list[i].LineEndPoint == list[j].LineStartPoint)

{

Line tmp = new Line();

tmp = list[i + 1];

list[i + 1] = list[j];

list[j] = tmp;

break;

}

else

{

Point Ptmp;

Ptmp=list[j].LineEndPoint;

list[j].LineEndPoint = list[j].LineStartPoint;

list[j].LineStartPoint = Ptmp;

Line tmp = new Line();

tmp = list[i + 1];

list[i + 1] = list[j];

list[j] = tmp;

break;

}

}

}

}

}

private bool IsLineAlone(Line line, Dictionary<int, Line> lines)

{

bool IsAlone=false;

bool IsStartAlone = true;

bool IsEndAlone = true;

List<Line> lines1 = new List<Line> ();

foreach(var item in lines)

{

lines1.Add(item.Value);

}

lines1.Remove(line);

foreach (var item in lines1)

{

if(item.LineEndPoint==line.LineEndPoint|| item.LineStartPoint == line.LineEndPoint)

{

IsEndAlone = false;

}

if (item.LineEndPoint == line.LineStartPoint || item.LineStartPoint == line.LineStartPoint)

{

IsStartAlone = false;

}

}

if(IsEndAlone|| IsStartAlone)

{

IsAlone = true;

}

return IsAlone;

}

private void PictureMouseMove(object sender, MouseEventArgs e)

{

if(CurrentLineClicked)

{

CurrentLine.LineEndPoint.X = e.X;

CurrentLine.LineEndPoint.Y = e.Y;

Point NearestP = NearestPoint(e.X, e.Y);

if(Math.Sqrt(Math.Pow((e.X - NearestP.X), 2) + Math.Pow((e.Y - NearestP.Y), 2)) <10)

{

CurrentLine.LineEndPoint.X = NearestP.X;

CurrentLine.LineEndPoint.Y = NearestP.Y;

}

pictureBox1.Invalidate();

}

}

private Point NearestPoint(double X,double Y)

{

Point NearestPoint=new Point();

double MinDistance = 1000;

foreach (var item in Lines)

{

if(Math.Sqrt(Math.Pow((X-item.LineStartPoint.X),2)+ Math.Pow((Y - item.LineStartPoint.Y), 2))<MinDistance)

{

MinDistance = Math.Sqrt(Math.Pow((X - item.LineStartPoint.X), 2) + Math.Pow((Y - item.LineStartPoint.Y), 2));

NearestPoint = item.LineStartPoint;

}

if (Math.Sqrt(Math.Pow((X - item.LineEndPoint.X), 2) + Math.Pow((Y - item.LineEndPoint.Y), 2)) < MinDistance)

{

MinDistance = Math.Sqrt(Math.Pow((X - item.LineEndPoint.X), 2) + Math.Pow((Y - item.LineEndPoint.Y), 2));

NearestPoint = item.LineEndPoint;

}

}

return NearestPoint;

}

private void ListBoxItemSelect(object sender, EventArgs e)

{

var g = pictureBox1.CreateGraphics();

Pen pen = new Pen(Color.Black);

Pen pen2 = new Pen(Color.Red);

foreach (var item in Shapes)

{

foreach(var line in item.LinesList)

{

g.DrawLine(pen, line.LineStartPoint, line.LineEndPoint);

}

}

if (listBox1.SelectedItems.Count != 0)

{

int SelectedIndex = listBox1.SelectedIndex;

textBox2.Text = Shapes[SelectedIndex].Square.ToString();

foreach (var line in Shapes[SelectedIndex].LinesList)

{

g.DrawLine(pen2, line.LineStartPoint, line.LineEndPoint);

}

}

}

private void CalcButtonClick(object sender, EventArgs e)

{

Form2 newForm = new Form2();

newForm.Show();

}

}

}

## **Текст программы дополнительного окна для вычисления площади по координатам**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace KPO\_Solomentsev

{

public partial class Form2 : Form

{

public Form2()

{

InitializeComponent();

listView1.Hide();

}

bool IsPressed=false;

int X;

int Y;

List<Point> points = new List<Point>();

private void ButtonClick(object sender, EventArgs e)

{

if(Int32.TryParse(textBox3.Text,out X)&& Int32.TryParse(textBox4.Text, out Y))

{

listView1.Show();

IsPressed = false;

Point NewPoint = new Point(X, Y);

if (points.Contains(NewPoint))

{

MessageBox.Show("Данная точка уже существует");

textBox3.Clear();

textBox4.Clear();

}

else

{

points.Add(NewPoint);

textBox3.Clear();

textBox4.Clear();

listView1.Items.Add("(" + NewPoint.X.ToString() + "; " + NewPoint.Y.ToString() + ")");

}

}

else

{

MessageBox.Show("Значения точек должны быть целочисленными");

textBox3.Clear();

textBox4.Clear();

}

}

private void CulcButtonClick(object sender, EventArgs e)

{

if(points.Count<3)

{

MessageBox.Show("Недостаточно точек для подсчета площади");

}

else

{

IsPressed = true;

pictureBox1.Image = null;

double Square = 0;

for (int i = 0; i < points.Count - 1; i++)

{

Square += points[i].X \* points[i + 1].Y;

Square -= points[i + 1].X \* points[i].Y;

}

Square += points[points.Count - 1].X \* points[0].Y - points[points.Count - 1].Y \* points[0].X;

Square = Math.Abs(Square);

Square /= 2.0;

MessageBox.Show("Площадь фигуры равна: " + Square.ToString());

Graphic();

}

}

private void Graphic()

{

var g = pictureBox1.CreateGraphics();

Pen pen = new Pen(Color.Black);

for (int i = 0; i < points.Count - 1; i++)

{

g.DrawLine(pen, points[i], points[i + 1]);

}

g.DrawLine(pen, points[points.Count - 1], points[0]);

}

private void PictureBoxPaint(object sender, PaintEventArgs e)

{

if(IsPressed)

{

Pen pen = new Pen(Color.Black);

for (int i = 0; i < points.Count - 1; i++)

{

e.Graphics.DrawLine(pen, points[i], points[i + 1]);

}

e.Graphics.DrawLine(pen, points[points.Count - 1], points[0]);

}

}

private void ClearButtonClick(object sender, EventArgs e)

{

IsPressed = false;

points.Clear();

listView1.Clear();

pictureBox1.Image = null;

}

private void XKeyDown(object sender, KeyEventArgs e)

{

if(e.KeyCode==Keys.Enter)

{

e.SuppressKeyPress = true;

if (textBox4.Text.Length == 0&& textBox3.Text.Length >0)

{

textBox4.Focus();

}

else

{

if(textBox4.Text.Length > 0 && textBox3.Text.Length > 0)

{

button1.PerformClick();

}

}

}

}

private void YKeyDown(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.Enter)

{

e.SuppressKeyPress = true;

if (textBox3.Text.Length == 0&& textBox4.Text.Length > 0)

{

textBox3.Focus();

}

else

{

if (textBox4.Text.Length > 0 && textBox3.Text.Length > 0)

{

button1.PerformClick();

}

}

}

}

}

}